Tektosyne User's Guide

Overview of Library Features by Namespace

Christoph Nahr
christoph.nahr@kynosarges.org

Published 09 June 2012 on the Tektosyne home page
http://www.kynosarges.org/Tektosyne.html

Abstract

This guide summarizes the contents of the Tektosyne Library for the .NET Framework. The current versions of this document, the library itself, and its class reference are available at the Tektosyne home page. Please see there for system requirements and other information.

This guide covers all Tektosyne namespaces in alphabetical order, grouping the types in each namespace by thematically related categories. The goal is to provide a compact overview of the library's functionality, while explaining some complex or unusual features in greater detail. Please see the *Tektosyne Class Reference* for a complete documentation of all types.

Online Reading. This document contains a "Bookmarks" navigation tree. Click on any tree node to jump to the corresponding section. Moreover, all phrases in blue color are clickable hyperlinks that will take you to the section or address they describe.

Colophon. This document was written in LaTeX using MiKTeX 2.9 with XeLaTeX, KOMA-Script, and various other packages. See LaTeX Typesetting with MiKTeX for details.

The UML diagrams were reverse-engineered from the compiled .NET assemblies, using my free Class Diagrammer application, and embedded as PDF files.

Body text is set in Minion 12 pt from Adobe's Minion Pro collection, designed by Robert Slimbach. Subtitles and diagram text are set in various sizes and weights of Myriad from Adobe's Myriad Pro collection, designed by Robert Slimbach and Carol Twombly.

Identifiers and code fragments outside of UML diagrams are set in Microsoft's Consolas, designed by Lucas de Groot. The font is artificially compressed by 20% to take up less space.

Date	Version	Library	Description
2012–06–09	1.2.0	5.6.3	Changed typesetting to LaTeX with MiKTeX
2012–05–30	1.1.1	5.6.3	Added `RectLocation`
2012–03–31	1.1.0	5.6.1	Changed typesetting to DITA with oXygen
2012–02–26	1.0.3	5.6.0	Added `VisualSource`, `ConcurrentVisualHost`
2012–01–09	1.0.2	5.5.6	Added `AssemblyExtensions`
2011–06–24	1.0.1	5.5.2	Updated `QuadTree<T>`, `Subdivision`, `IGraph2D<T>`
2011–05–31	1.0.0	5.5.1	Initial release, using DITA with FrameMaker

Contents

List of Figures

Chapter 1

Assemblies

The Tektosyne Library ships in two assemblies. `Tektosyne.Core` only requires the four "core" BCL assemblies available to a Portable Class Library, whereas `Tektosyne.Windows` requires many other BCL assemblies that are specific to Windows. This separation should allow most Tektosyne features to work on any platform that supports the .NET Framework 4.0, including Mono, XNA, and Windows Phone 7. Please see the library's `ReadMe` file for further details on subject.

The distribution of namespaces across assemblies is shown in Figure 1.1. Two namespaces are present in both assemblies: `Tektosyne.Geometry` is generally platform-independent but also provides some conversions to GDI+ and WPF, and the split of `Tektosyne.Xml` was necessary because some XML APIs are not supported by the four "core" assemblies. The following chapters will note which assembly contains each type.

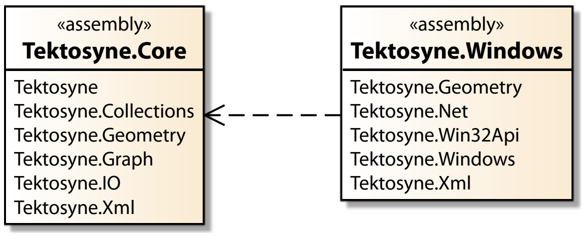

Figure 1.1: Assembly Namespaces

The Tektosyne download package also contains an assembly with unit tests for NUnit and a GUI application with testing and demonstration dialogs, `Tektosyne.GuiTest.exe`. This guide does not cover these assemblies, but Windows users are encouraged to explore the `GuiTest` application.

Chapter 2

Root Namespace

The root namespace `Tektosyne` contains types related to the BCL root namespace `System`, or at least unrelated to any of the more specific namespaces. Most can be grouped in thematically related sections, except for the following two types.

`AssemblyExtensions` — Provides extension methods to conveniently retrieve informational attributes and public key tokens from a given `Assembly`.

`EventArgs<T>` — Provides an event argument that contains a single arbitrary value, transmitting data to an event handler without having to derive a dedicated type from `System.EventArgs`.

2.1 Exceptions

This section comprises several specialized exception types, as well as numerous helper methods to simplify throwing exceptions.

`ArgumentNullOrEmptyException` — Derives from `System.ArgumentException`. Thrown when an argument is either a null reference or "empty" in some type-specific way, e. g. an empty string.

`AssertionException` — Thrown by the `ThrowHelper.Assert` method. Useful when you wish to avoid the standard `Debug/Trace.Assert` mechanism and throw a regular exception instead.

`DetailException` — Adds a second text property for technical details. Related features include method `Tektosyne.Xml.XmlUtility.GetXmlMessage` to construct the detail text, and WPF dialog `Tektosyne.Windows.MessageDialog` to show both text properties.

`PropertyValueException` — Derives from `System.InvalidOperationException`. Thrown when an operation fails because a property has an invalid value (the usual reason for "invalid operations").

`ThrowHelper` — Static methods that create and throw many common exceptions. This simplifies string formatting and reduces the caller's memory footprint compared to the surprisingly large amount of machine code produced for literal `throw` instructions. For short methods, this size difference can determine whether the method is inlined by the JIT compiler.

2.2 Mathematics

This section comprises several mathematical types and methods unrelated to computational geometry; see the namespace `Tektosyne.Geometry` for the latter.

`Fortran` — Static methods whose names and semantics mirror standard functions of Fortran 90, with overloads for all numerical types except for `System.Complex`.

`MathUtility` — Static methods providing epsilon comparisons, prime number tests, and range restriction and normalization.

`MersenneTwister` — Pseudo-random number generator MT19937, known as "Mersenne Twister." The C# implementation is based on the original C program by Takuji Nishimura and Makoto Matsumoto, with additions by Isaku Wada.

2.3 Strings

This section comprises the library's localizable string resources, as well as several types and methods for string manipulation.

`Strings` — Contains all localizable strings of the Tektosyne library, including many general-purpose error messages that may be useful in other applications.

`StringUtility` — Static methods that provide natural string sorting (i. e. numerical substrings are sorted numerically), testing strings for RTF text and e-mail addresses, and more.

`NaturalString` — Combines a `System.String` with implicit natural string sorting, using the sorting rules of the current culture for non-numerical characters.

`OrdinalString` — Combines a `System.String` with implicit natural string sorting, using ordinal sorting rules for non-numerical characters.

2.4 Tuples

.NET 4.0 defines a family of generic `System.Tuple` types, intended to support the new functional programming language F#. All standard tuples are immutable classes: immutable because F# prefers immutable objects, and classes because they can grow to any size.

Although C# lacks built-in support for tuples, they are still useful to group several values without having to define a dedicated type. With this purpose in mind, the root namespace provides two alternative tuple implementations. Both flavors define instances for 2–4 elements, and static factory methods that allow generic type inference.

MutableTuple<T_1,…,T_n> — Generic tuples implemented as *mutable* classes (i. e. reference types). Not usually desirable, but convenient in special cases such as two-way data binding to WPF controls.

ValueTuple<T_1,…,T_n> — Generic tuples implemented as immutable *structures* (i. e. value types). More efficient for small amounts of data, such as combining a few numerical and boolean values.

Chapter 3

Collections Namespace

The namespace `Tektosyne.Collections` contains a variety of generic collections, some of which extend standard collections in the BCL namespace `System.Collections.Generic`, as well as helper types and methods for manipulating collections.

Note. To save space, generic type parameters that represent dictionary keys and values are shown as `K` and `V`, instead of the actual `TKey` and `TValue` used by the library.

3.1 Collection Keys

Standard dictionaries associate elements with separate keys. In practice, however, many element types also embed the same key in their own data, e. g. as an `Id` or `Name` property. This causes two problems: the associated key might not match the embedded key, and the separately stored dictionary key wastes memory.

The `IKeyedValue<K>` interface tackles both issues with a `Key` property that returns the object's embedded key. Tektosyne collections that store key-and-value pairs require or allow `IKeyedValue<K>` instances for the values, and check for matching associated and embedded keys. Other Tektosyne collections store lists of `IKeyedValue<K>` instances that can be treated like dictionaries while using only embedded keys.

`IKeyedValue<K>`	— Associates an object with a key, exposed as a read-only `Key` property.
`IMutableKeyedValue<K>`	— Adds a `SetKey` method to the `IKeyedValue<K>` interface, allowing key changes.
`KeyMismatchException`	— Thrown when a Tektosyne collection attempts to store an `IKeyedValue<K>` object with an associated key that differs from its embedded `Key` property.

3.2 Extended Collections

The collections with an …Ex suffix provide extended versions of standard collections with a variety of extra features. QueueEx<T> and StackEx<T> derive directly from the corresponding standard classes, whereas the other extended collections use the standard classes for their internal backing storage, with standard methods reimplemented as forwarders.

The reason for this is that all extended collections, except for QueueEx<T> and StackEx<T>, provide a *read-only wrapper* as their most significant new feature. Like the AsReadOnly methods of the standard classes Array and List<T>, the wrapper is a "live" view that reflects all changes made to the underlying collection and requires no memory for content duplication.

However, our implementation returns the same type as the underlying collection rather than a standard ReadOnlyCollection<T>, allowing the transparent substitution of read-only wrappers for writable collections at runtime. The read-only wrapper is cached with each collection, and a public SyncRoot property refers both the writable collection and its read-only wrapper (if any) to the SyncRoot of their shared backing storage for multithreaded synchronization.

Moreover, a static read-only Empty field holds the read-only wrapper of an empty collection. Since the Empty collection is immutable, it is a safe and efficient alternative to null references for representing collections that cannot have any elements.

Other additions include a Copy method that performs a deep copy if all elements implement ICloneable, an Equals method that compares two collections for equal contents, automatic key checking for dictionaries that store IKeyedValue<K> elements, and more.

ArrayEx<T> — Provides an extended Array – that is, a fixed-size collection – that can be indexed in one or more dimensions, according to a user-defined rank. Various methods convert indices and array contents between their one- and multi-dimensional representations.

DictionaryEx<K,V> — Provides an extended Dictionary<K,V>, i. e. a dynamic hashtable.

ListEx<T> — Provides an extended List<T> which optionally ensures that all elements are unique.

QueueEx<T> — Provides an extended Queue<T>, i. e. a first-in, first-out (FIFO) collection.

SortedDictionaryEx<K,V> — Provides an extended SortedDictionary<K,V>, i. e. a red-black binary search tree.

SortedListEx<K,V> — Provides an extended SortedList<K,V> with the very useful methods GetByIndex, GetKey, and SetByIndex that were present in the standard non-generic SortedList, but inexplicably dropped from the generic version.

StackEx<T> — Provides an extended Stack<T>, i. e. a last-in, first-out (LIFO) collection.

3.3 Specialized Collections

The following collections specialize other generic collection classes. All of them implement the common extensions described in the previous section.

The first two collections replace a generic key type with `Int32`. Integer values are their own hash codes and can be compared in a single machine instruction, obviating the need for slow calls to generic hashing and comparison methods.

The remaining collections derive from `ListEx<T>` to provide dictionaries with an unsorted indexed backing storage, maintaining the insertion order of elements. Key and index access mimics the pattern defined by `SortedListEx<K,V>`.

`Int32Dictionary<V>` — Provides a `Dictionary<K,V>` whose keys are `Int32` values.

`Int32HashSet` — Provides a limited `HashSet<T>` whose elements are `Int32` values.

`KeyedList<K,V>` — Provides a `ListEx<V>` whose elements are accessible by embedded keys. All `V` elements must implement `IKeyedValue<K>`.

`KeyValueList<K,V>` — Provides a `ListEx<KeyValuePair<K,V>>` whose elements are key-and-value pairs that are accessible by their key components.

`MultiKeyedList<K,V>` — Provides a `ListEx<V>` whose elements are accessible by keys that are derived from each element by an arbitrary key converter. The converter method can be changed after construction, allowing one collection to support multiple keys over its lifetime.

3.4 Tree Collections

This section comprises two popular search trees absent from the BCL. The C# implementations are based on Michael J. Laszlo's *Computational Geometry and Computer Graphics in C++*, Prentice Hall 1996. (The braided search tree is not inherently a geometric data structure, but it is well-suited for geometric sweep line algorithms.)

Both trees expose their nodes as public classes with read-only properties. This allows clients to directly navigate the tree structure, including all internal nodes, rather than merely performing searches or enumerating leaves

`BraidedTree<K,V>` — Provides a generic braided search tree, i. e. a binary search tree whose nodes are connected in key sorting order by a doubly-linked list. Finding the smallest, largest, next-smallest, and next-largest element are all O(1) operations. The tree uses randomized balancing rather than the red-black balancing of the standard `SortedDictionary<K,V>`.

`BraidedTreeNode<K,V>` — Provides a node in a `BraidedTree<K,V>`.

QuadTree<V> — Provides a generic quadrant tree whose keys are `Tektosyne.Geometry.PointD` values, i. e. a two-dimensional search tree that recursively divides a specified bounding rectangle into equal-sized quadrants. Finding the quadrant that contains a given point and finding all points within a given range are both logarithmic operations.

Notable features include a heuristic depth probe to speed up searches in large trees, inspired by Sariel Har-Peled's lecture *Quadtrees – Hierarchical Grids;* and a `Move` method that can reduce successive key changes to O(1) operations, provided that old and new keys are clustered within the same leaf node.

QuadTreeNode<V> — Provides a node in a `QuadTree<V>`.

3.5 Comparing Objects

The BCL provides several incompatible ways to compare two arbitrary objects: a non-generic and a generic interface, and a generic delegate. Since delegates are the most basic representation, two simple adapter classes wrap them in the equivalent interfaces.

ComparerAdapter<T> — Wraps a `Comparison<T>` method in the `IComparer` and `IComparer<T>` interfaces, for compatibility with consumers that require these interfaces.

ComparerCache<T> — Caches `(Equality)Comparer<T>.Default` for faster access. These BCL properties already perform their own caching, but access is still so slow that the extra cache level provides a noticeable speedup.

EqualityComparerAdapter<T> — Wraps an `Equals<T,T>` method in the `IEqualityComparer` and `IEqualityComparer<T>` interfaces, for compatibility with consumers that require these interfaces.

3.6 Sorting & Searching

This section comprises numerous helper methods and standard sorting & searching algorithms for arbitrary collections, typed as generic or non-generic standard interfaces.

CollectionsUtility — Helper methods for comparing collections; creating index arrays; moving and swapping elements; selecting a random element and randomizing element order; and manipulating collections with `IKeyedValue<T>` and `IMutableKeyedValue<T>` elements.

Sorting — Standard sorting algorithms for arbitrary `IList<T>` collections, as well as a binary search for sorted `IList<T>` collections. The sorting algorithms include heap sort, insertion sort, quick sort, and shell sort. The

C# implementations are based on Robert Sedgewick's *Algorithms in Java* (3rd ed.), Addison-Wesley 2003.

The quick sort and binary search algorithms are slower than the highly optimized BCL algorithms, but the latter only work on arrays and array-backed standard collections. Two additional methods, `BestQuickSort` and `BestBinarySearch`, dispatch to the superior BCL algorithms if the concrete specified collection supports them.

Chapter 4

Geometry Namespace

The namespace `Tektosyne.Geometry` covers the field of computational geometry, including a set of geometric primitives as well as a variety of standard algorithms and data structures. All types use two-dimensional coordinates exclusively.

Many algorithms were adapted from C/C++ and pseudocode programs in standard literature, including the following sources. Please consult the *Class Reference* for details.

— Mark de Berg et al., *Computational Geometry,* Springer-Verlag 2008 (3rd ed.)

— Michael J. Laszlo, *Computational Geometry and Computer Graphics in C++,* Prentice Hall 1996

— Joseph O'Rourke, *Computational Geometry in C,* Cambridge University Press 1988 (2nd ed.)

The orientation of the vertical axis is somewhat problematic in computational geometry. The standard mathematical orientation has y-coordinates increase upward, but the standard drawing orientation of computer graphics puts the origin in the upper-left corner of the screen and has y-coordinates increase downward.

The *Tektosyne Class Reference* notes the actual orientation wherever it is relevant. Generally, `Tektosyne.Geometry` types assume mathematical orientation. One major exception are the rectangle primitives whose `Top` property refers to the *smallest* y-coordinate, for compatibility with BCL rectangles that were designed for screen display.

Another frequent source of trouble is floating-point imprecision. Some `Tektosyne.Geometry` algorithms use a fixed comparison epsilon of 1^{-10} to achieve numerical stability, while others allow a user-defined epsilon. Some algorithms are available in both exact and epsilon variants. You need to experiment with your own data to determine the most suitable variant.

When an algorithm accepts a user-defined epsilon, you can usually choose a fairly large value that merges clearly distinct points rather than just compensating for floating-point imprecision. One application is to map the location of a user's mouse click on the screen to a nearby point in a geometric data structure. The demo application `Tektosyne.GuiTest.exe` offers several test dialogs that let you experiment with super-sized comparison epsilons.

4.1 Geometric Primitives

The BCL namespaces `System.Drawing` and `System.Windows` each provide a set of geometric primitives in two dimensions (points, sizes, lines, rectangles). Unfortunately, both require Windows-specific assemblies and are mutually incompatible to boot. Therefore, we define our own geometric primitives which compare to the BCL types as follows:

— All types are immutable structures, as one would expect. Strangely, the BCL types are mutable structures, perhaps for compatibility with Visual Studio's GUI designers.

— All types define a static read-only field `Empty` that holds a default-initialized instance, as with the `System.Drawing` types. The `System.Windows` types redefine `Empty` to hold a "magic" invalid value, which is confusing and a bad idea to begin with. Use `Nullable<T>` or a separate boolean flag to represent invalid values.

— Points define addition and subtraction with other points; sizes with other sizes. The BCL offers an addition of points and sizes instead, which I found rather useless.

— Sizes and rectangles always require a non-negative width and height. Some of the BCL sizes and rectangles also perform these checks while others do not.

— Integer rectangles *exclude* their greatest coordinates in both dimensions whereas floating-point rectangles *include* them. This reproduces the behavior of BCL types which stems from the old GDI habit of using integer extensions to control C-style `for` loops.

— There is no dedicated vector type. The BCL type `System.Windows.Vector` holds exactly the same data as `System.Windows.Point`, and so merely forces developers to convert point coordinates back and forth in order to use vector operations. Instead, our point types directly implement the features of `System.Windows.Vector`.

All geometric primitives are available with the three coordinate types `Double`, `Single`, and `Int32`, indicated by a `D`, `F`, and `I` suffix, respectively. The `Double` and `Single` variants are fully equivalent whereas the `Int32` variants lack some algorithms that make no sense for integer coordinates.

Instance methods on geometric primitives generally operate with the same precision used to represent coordinates, except for methods on `Int32` primitives which use `Double` precision when fractional results are expected. The stand-alone algorithms described in the following sections always expect `Double` coordinates and operate with `Double` precision.

`LineD, LineF, LineI`	— Provide line segments with `Double`, `Single`, and `Int32` coordinates.
`PointD, PointF, PointI`	— Provide spatial locations with `Double`, `Single`, and `Int32` coordinates.

`RectD, RectF, RectI`	— Provide rectangles with `Double`, `Single`, and `Int32` coordinates.
`SizeD, SizeF, SizeI`	— Provide spatial extensions with `Double`, `Single`, and `Int32` coordinates.

4.2 Basic Algorithms

This section comprises basic constants and algorithms for computational geometry which do not fall in any of the more specialized categories covered below.

Some basic algorithms are defined as instance methods on geometric primitives. These include angle calculations and vector operations on `PointD/F/I`; locating a point relative to a line segment on `LineD/F/I`; and the Liang-Barsky line clipping algorithm on `RectD/F`.

`Angle`	— Constants and methods to convert, normalize, and compare angles.
`Compass`	— Specifies the eight major compass directions as angles in degrees, starting with zero degrees for north and continuing clockwise.
`GeoAlgorithms`	— Static methods for creating random points, line segments, and polygons; computing the area and centroid of a polygon; computing the convex hull of a point set (Graham scan); locating a point relative to a polygon (ray crossings algorithm); and more.
`LineLocation`	— Specifies the location of a point relative to a line segment: on or before the start point, on or after the end point, between both points, or to the left or right of the line.
`PolygonLocation`	— Specifies the location of a point relative to a polygon: strictly inside, strictly outside, or coinciding with an edge or a vertex.
`RectLocation`	— Specifies the location of a point relative to a rectangle, expressed as a bitwise combination of each point coordinate's location relative to the rectangle extension in the same dimension.

4.3 Line Intersection

Several algorithms intersect two or more line segments, represented either by `LineD` instances or pairs of `PointD` coordinates. The `LineD/F/I` structures also define instance methods that forward to the two-segment algorithm.

`LineIntersection`	— Defines a robust algorithm for finding the intersection, if any, between two line segments or their infinite extensions, and also holds the result of the algorithm.

The algorithm examines both the cross-product lengths for each triplet of end points and the line equation parameters for both segments to determine intermediate results. If the test results contradict each other, the algorithm starts over with a greater comparison epsilon until both tests agree. The minimum comparison epsilon is always 1^{-10} to avoid such recursions in most cases.

`LineRelation` — Specifies the relationship between two line segments: parallel, collinear, or divergent.

`MultiLineIntersection` — Defines both a brute-force and a sweep line algorithm for finding all points of intersection between multiple line segments. The brute-force algorithm simply intersects all input lines with each other. This is always an $O(n^2)$ operation but has virtually no overhead and can accept a comparison epsilon greater than 1^{-10} to merge nearby crossings.

The Bentley-Ottmann sweep line algorithm is faster for large input sets with few intersections, but otherwise slower due to its large overhead. An improved sweep line comparer raises numerical stability to the level of the brute-force algorithm.

`MultiLinePoint` — Contains the results of either `MultiLineIntersection` algorithm.

4.4 Point Comparisons

Two `IComparer<PointD>` implementations compare points lexicographically, preferring either the x- or the y-coordinate, and an interface exposes their common features. Comparisons can be performed exactly or with a user-defined epsilon.

Both classes provide a nearest-point search for lexicographically sorted `PointD` collections. The algorithm first performs a binary search in the preferred dimension to approximate the index of the search coordinates, and then expands a radius around that index until the nearest point is found. This heuristic can achieve a runtime of $O(\mathrm{ld}\ n)$ with no additional overhead, assuming that coordinates are more or less evenly distributed throughout the collection.

`IPointDComparer` — Compares two `PointD` instances lexicographically. The comparison order depends on the concrete implementation.

`PointDComparerX` — Compares two `PointD` instances lexicographically, preferring x-coordinates.

`PointDComparerY` — Compares two `PointD` instances lexicographically, preferring y-coordinates.

4.5 Regular Polygons

The following types provide a flexible and efficient representation of regular polygon grids. The customizable maps of the Hexkit Strategy Game System are based on these types, and the *Hexkit*

User's Guide describes them in greater detail. The demo application `Tektosyne.GuiTest.exe` also provides a dialog to save and print arbitrary polygon grids.

`PolygonGrid` — Provides a rectangular grid of regular polygons with two-dimensional indexing. Features include efficient distance calculations, conversions between grid and display coordinates, a read-only wrapper similar to those of `Tektosyne.Collections` classes, and pathfinding between grid locations using `Tektosyne.Graph` algorithms.

`PolygonGrid.SubdivisionMap` — Maps the elements of a `PolygonGrid` to the faces of an equivalent `Subdivision`. This conversion is only intended for testing, as `PolygonGrid` is far more efficient than `Subdivision`.

`PolygonGridShift` — Specifies the shifting of rows or columns in a `PolygonGrid`, i. e. whether the second row or column is shifted right or down compared to the first one.

`RegularPolygon` — Provides a regular polygon with three or more sides. A `RegularPolygon` with four or six sides can be used to construct a `PolygonGrid`.

`PolygonOrientation` — Specifies the orientation of a `RegularPolygon`: lying on an edge or standing on a vertex.

4.6 Planar Subdivisions

The following types represent a planar subdivision, i. e. any collection of line segments that intersect only at their end points, as a doubly-connected edge list (DCEL). This representation is memory-intensive but allows fast navigation through all elements of the subdivision.

Since any planar graph with straight bounded edges can be represented as a `Subdivision`, several other types provide conversions to this class, including `PolygonGrid` and `Voronoi`. A dedicated interface maps the resulting `Subdivision` faces to elements of the original structure.

`Subdivision` — Provides a planar subdivision composed of straight bounded edges, vertices on the end points of edges, and faces formed by closed loops of edges. Pathfinding between vertices is supported by `Tektosyne.Graph` algorithms.

You can create a new `Subdivision` from a given set of line segments or polygons, or by intersecting two existing instances. You can also add or remove individual edges, split edges in half, and move or delete individual vertices (along with their edges). This set of operations allows interactive editing of a `Subdivision`.

`SubdivisionEdge` — Provides one half-edge in a `Subdivision`. Half-edges are always paired with twin half-edges in the opposite direction to form one full edge of the planar subdivision.

`SubdivisionFace` — Provides one face in a `Subdivision`. Faces are polygons that may or may not enclose any area. Faces with a positive area may contain one or more "holes," i. e. interior faces. Every subdivision also contains one unbounded face that represents the entire two-dimensional plane and thus encloses all bounded faces as its "holes."

`SubdivisionSearch` — Provides a fast but memory-intensive search structure for a `Subdivision`. The `Subdivision` class itself provides slower brute-force searches that require no additional memory.

`SubdivisionElement` — Represents an arbitrary `Subdivision` element, i. e. one vertex, half-edge, or face. Returned by the general search algorithms `Subdivision.Find` and `SubdivisionSearch.Find`.

`SubdivisionElementType` — Specifies the type of a `SubdivisionElement`: vertex, half-edge, or face.

`ISubdivisionMap<T>` — Provides a bidirectional mapping between the faces of a `Subdivision` and the generically typed objects in some arbitrary collection.
Creating a `Subdivision` from another geometric structure automatically establishes a structural mapping. Applications might also define semantic mappings, e. g. correlating statistical information to faces that represent geographical areas.

See Figure 4.1 for the public interfaces of all types listed above, as well as their mutual relationships (with some elided for lack of space). The implementation follows the DCEL structure outlined by Mark de Berg et al., *Computational Geometry,* Springer-Verlag 2008 (3rd ed.), but a few peculiar features are worth pointing out.

4.6.1 Edge and Face Keys

Every `SubdivisionEdge` and `SubdivisionFace` is identified by an `Int32` key that is unique within its `Subdivision`. The corresponding `Edges` and `Faces` collections provide an O(ld n) access by key, or an O(1) access by index if already known.

The ascending sequence of keys reflects the order in which the `Subdivision` was constructed. Keys are normally immutable but can be renumbered to plug "holes" in the sequence caused by dynamic edge deletion.

Strictly speaking, these keys are an unnecessary feature. References and/or indices would suffice to identify half-edges and faces. However, keys so enormously simplify unit testing and debugging that they are worth the extra memory.

4.6.2 Half-Edge Cycles

The half-edge cycle that contains a `SubdivisionEdge` constitutes a polygon which may have a positive area. `CyclePolygon` builds the polygon, but several other properties avoid this step and

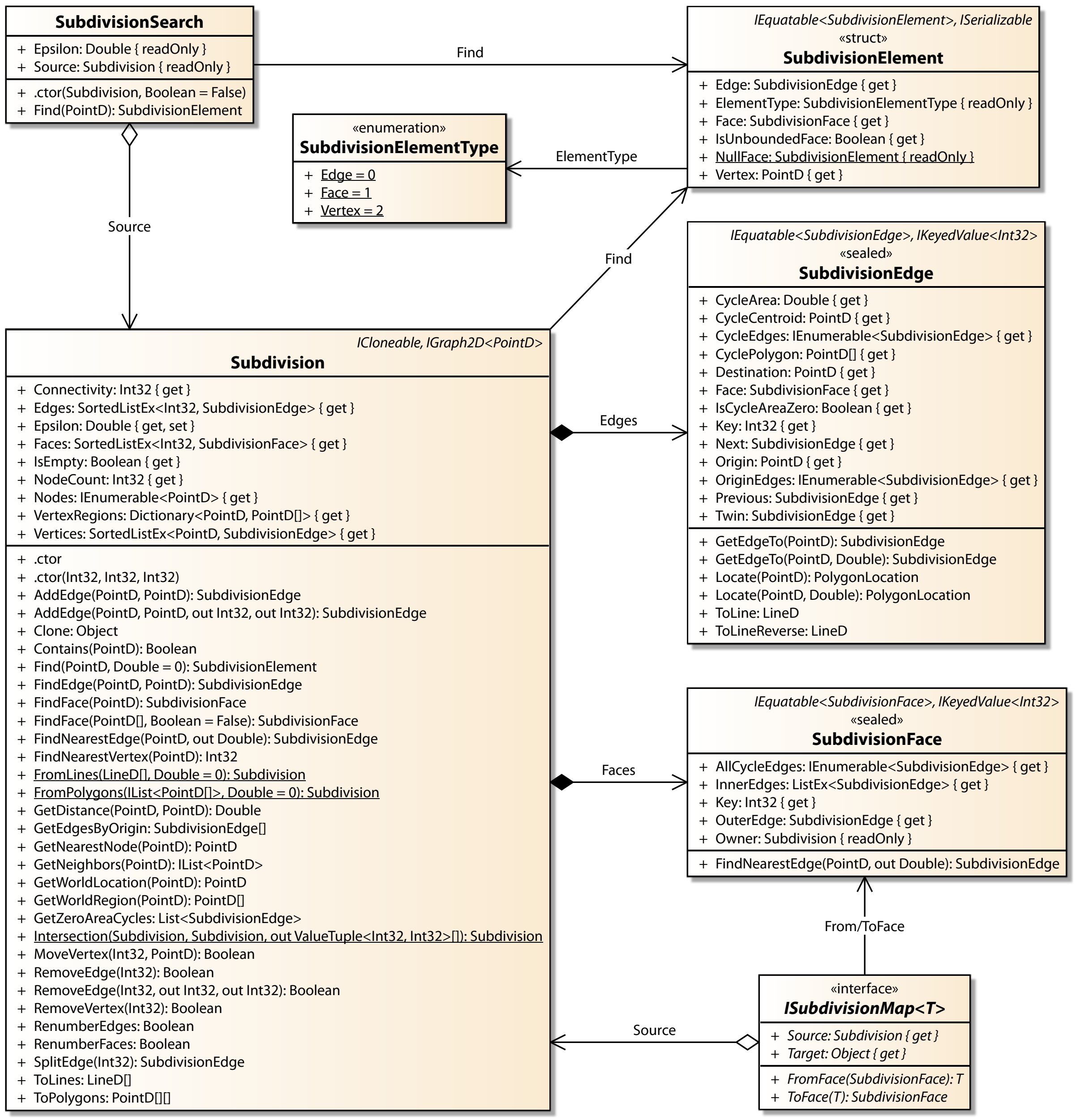
SubdivisionSearch
+ Epsilon: Double { readOnly }
+ Source: Subdivision { readOnly }
+ .ctor(Subdivision, Boolean = False)
+ Find(PointD): SubdivisionElement
Find
IEquatable<SubdivisionElement>, ISerializable
«struct»
SubdivisionElement
+ Edge: SubdivisionEdge { get }
+ ElementType: SubdivisionElementType { readOnly }
+ Face: SubdivisionFace { get }
+ IsUnboundedFace: Boolean { get }
+ NullFace: SubdivisionElement { readOnly }
+ Vertex: PointD { get }
«enumeration»
SubdivisionElementType
+ Edge = 0
+ Face = 1
+ Vertex = 2
ElementType
Source
Find
IEquatable<SubdivisionEdge>, IKeyedValue<Int32>
«sealed»
SubdivisionEdge
+ CycleArea: Double { get }
+ CycleCentroid: PointD { get }
+ CycleEdges: IEnumerable<SubdivisionEdge> { get }
+ CyclePolygon: PointD[] { get }
+ Destination: PointD { get }
+ Face: SubdivisionFace { get }
+ IsCycleAreaZero: Boolean { get }
+ Key: Int32 { get }
+ Next: SubdivisionEdge { get }
+ Origin: PointD { get }
+ OriginEdges: IEnumerable<SubdivisionEdge> { get }
+ Previous: SubdivisionEdge { get }
+ Twin: SubdivisionEdge { get }
+ GetEdgeTo(PointD): SubdivisionEdge
+ GetEdgeTo(PointD, Double): SubdivisionEdge
+ Locate(PointD): PolygonLocation
+ Locate(PointD, Double): PolygonLocation
+ ToLine: LineD
+ ToLineReverse: LineD
ICloneable, IGraph2D<PointD>
Subdivision
+ Connectivity: Int32 { get }
+ Edges: SortedListEx<Int32, SubdivisionEdge> { get }
+ Epsilon: Double { get, set }
+ Faces: SortedListEx<Int32, SubdivisionFace> { get }
+ IsEmpty: Boolean { get }
+ NodeCount: Int32 { get }
+ Nodes: IEnumerable<PointD> { get }
+ VertexRegions: Dictionary<PointD, PointD[]> { get }
+ Vertices: SortedListEx<PointD, SubdivisionEdge> { get }
+ .ctor
+ .ctor(Int32, Int32, Int32)
+ AddEdge(PointD, PointD): SubdivisionEdge
+ AddEdge(PointD, PointD, out Int32, out Int32): SubdivisionEdge
+ Clone: Object
+ Contains(PointD): Boolean
+ Find(PointD, Double = 0): SubdivisionElement
+ FindEdge(PointD, PointD): SubdivisionEdge
+ FindFace(PointD): SubdivisionFace
+ FindFace(PointD[], Boolean = False): SubdivisionFace
+ FindNearestEdge(PointD, out Double): SubdivisionEdge
+ FindNearestVertex(PointD): Int32
+ FromLines(LineD[], Double = 0): Subdivision
+ FromPolygons(IList<PointD[]>, Double = 0): Subdivision
+ GetDistance(PointD, PointD): Double
+ GetEdgesByOrigin: SubdivisionEdge[]
+ GetNearestNode(PointD): PointD
+ GetNeighbors(PointD): IList<PointD>
+ GetWorldLocation(PointD): PointD
+ GetWorldRegion(PointD): PointD[]
+ GetZeroAreaCycles: List<SubdivisionEdge>
+ Intersection(Subdivision, Subdivision, out ValueTuple<Int32, Int32>[]): Subdivision
+ MoveVertex(Int32, PointD): Boolean
+ RemoveEdge(Int32): Boolean
+ RemoveEdge(Int32, out Int32, out Int32): Boolean
+ RemoveVertex(Int32): Boolean
+ RenumberEdges: Boolean
+ RenumberFaces: Boolean
+ SplitEdge(Int32): SubdivisionEdge
+ ToLines: LineD[]
+ ToPolygons: PointD[][]
Edges
Faces
IEquatable<SubdivisionFace>, IKeyedValue<Int32>
«sealed»
SubdivisionFace
+ AllCycleEdges: IEnumerable<SubdivisionEdge> { get }
+ InnerEdges: ListEx<SubdivisionEdge> { get }
+ Key: Int32 { get }
+ OuterEdge: SubdivisionEdge { get }
+ Owner: Subdivision { readOnly }
+ FindNearestEdge(PointD, out Double): SubdivisionEdge
From/ToFace
«interface»
ISubdivisionMap<T>
+ Source: Subdivision { get }
+ Target: Object { get }
+ FromFace(SubdivisionFace): T
+ ToFace(T): SubdivisionFace
Source

Figure 4.1: Subdivision Types

directly operate on the linked half-edges. `CycleArea` and `CycleCentroid` calculate the polygon's area and centroid, respectively, and `IsCycleAreaZero` tests for a zero-area polygon by examining face pointers rather than the area itself, which also avoids rounding errors.

`CycleEdges` employs the C# enumerator state machine to enumerate all half-edges in the cycle. `OriginEdges` does the same for all half-edges emanating from the same `Origin`, and `SubdivisionFace.AllCycleEdges` for all half-edge cycles within the same face. These properties are convenient rather than efficient, due to the overhead of the state machine. Use explicit `do-while` loops over the linked half-edges for maximum performance.

4.6.3 Vertex Distances

The `IGraph2D<T>` method `GetDistance` returns the actual Euclidean distance between vertices, including the final square root, rather than the less expensive squared distance. This is necessary to avoid overestimating the total cost of compound paths within the subdivision.

Assume a straight path consisting of multiple edges so that the total Euclidean distance equals the sum of the lengths of all edges. If `GetDistance` returned a squared Euclidean distance, the sum of all edge results would be smaller than the result for the two extreme vertices. This violates the invariant that the sum of the distances between all successive nodes within a sequence is never less than the distance between any two nodes from the same sequence.

4.6.4 Vertex Regions

The `VertexRegions` collection can associate `Vertices` with user-defined polygonal regions. The `IGraph2D<T>` method `GetWorldRegion` returns elements from this collection. The elements must be set explicitly, as the vertices of an arbitrary subdivision imply no meaningful regions.

As a typical example, you might create the `Subdivision` from a Delaunay triangulation and assign the Voronoi regions of its dual graph to the `VertexRegions` collection.

4.7 Voronoi Diagrams

The following types construct two standard structures from a given set of generator sites: the Voronoi diagram, whose polygonal regions comprise all points that are nearest to each generator site; and the Delaunay triangulation, its dual graph, whose edges are the nearest-neighbor connections for all generator sites.

`Voronoi`	— Defines a sweep line algorithm to find the Delaunay triangulation and (optionally) the Voronoi diagram for a given `PointD` set, in logarithmic time. The C# implementation is based on the original C program by Steven J. Fortune.
`VoronoiEdge`	— Represents one edge in a `Voronoi` diagram, including the diagram vertices that terminate the edge as well as the generator sites that are bisected by the edge.
`VoronoiResults`	— Contains the results of the `Voronoi` algorithm, and converts them to other representations. This includes a `Subdivision` based on the Voronoi diagram itself, and another based on the corresponding Delaunay triangulation.

`VoronoiResults.SubdivisionMap` — Maps the regions and generator sites of a `Voronoi` diagram to the faces of an equivalent `Subdivision`. Note that pathfinding between generator sites requires the `Subdivision` for the corresponding Delaunay triangulation, as the pathfinding algorithms in namespace `Tektosyne.Graph` operate only on the edges of a planar subdivision.

4.8 Windows Specifics

Several static helper classes reference BCL types from the `System.Drawing` and `System.Windows` namespaces. Since these libraries are specific to the Microsoft Windows platform, the following types reside in assembly `Tektosyne.Windows`.

`GdiConversions` — Converts between geometric primitives and equivalent `System.Drawing` types.

`PolygonExtensions` — Renders `RegularPolygon` and `PolygonGrid` instances to `System.Windows` types.

`WpfConversions` — Converts between geometric primitives and equivalent `System.Windows` types.

Chapter 5

Graph Namespace

The namespace Tektosyne.Graph provides four general-purpose graph algorithms that operate on two generic interfaces. Figure 5.1 shows all nine types in the namespace, as well as the two Tektosyne.Geometry types that implement the central graph interface.

Since the mechanisms implemented here are rather complex and not based on any well-known standards, this chapter goes into greater detail than usual and examines individual members where appropriate. To see the graph algorithms in action, try the following:

— The demo application Tektosyne.GuiTest.exe contains an interactive test that runs all four algorithms on both PolygonGrid and Subdivision graphs.

— The Hexkit Strategy Game System utilizes a complex customizable implementation based on PolygonGrid graphs. The *Hexkit User's Guide* also describes the interaction of the game system and the pathfinding mechanisms.

5.1 Graphs and Agents

The two basic interfaces that connect the four generic algorithms with custom applications are IGraph2D<T> and IGraphAgent<T>. The first represents the graph itself on which searches take place, and must always be implemented. The second represents some mobile agent that traverses the graph, and is required for the AStar<T> and Coverage<T> algorithms.

5.1.1 Graph Structure

The central interface IGraph2D<T> represents any graph whose T nodes map to polygonal regions in two-dimensional space. All graph algorithms are created with an IGraph2D<T> instance on which all searches are performed.

The namespace Tektosyne.Geometry contains two graph implementations, PolygonGrid and Subdivision. The PolygonGrid node type is PointI: each graph node is the two-dimensional

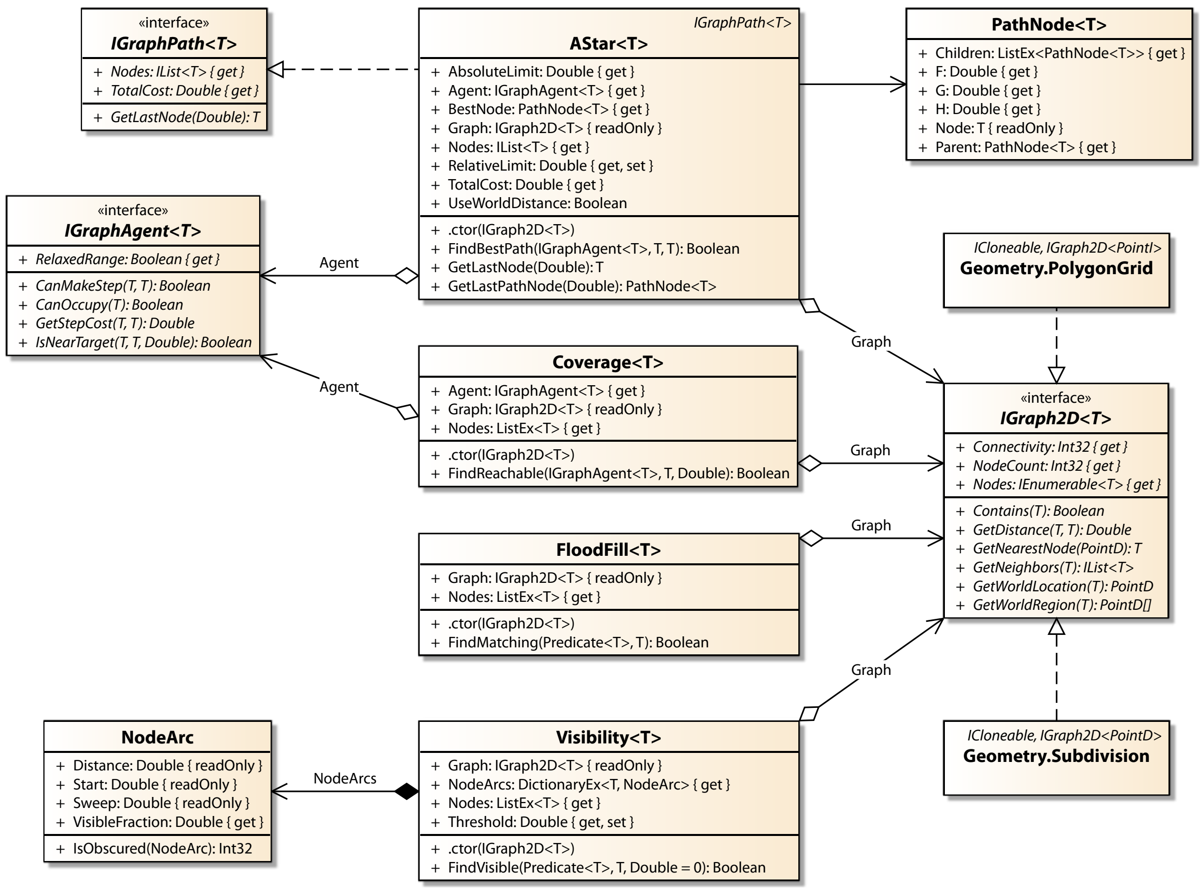
«interface»
IGraphPath<T>
+ Nodes: IList<T> { get }
+ TotalCost: Double { get }
+ GetLastNode(Double): T
IGraphPath<T>
AStar<T>
+ AbsoluteLimit: Double { get }
+ Agent: IGraphAgent<T> { get }
+ BestNode: PathNode<T> { get }
+ Graph: IGraph2D<T> { readOnly }
+ Nodes: IList<T> { get }
+ RelativeLimit: Double { get, set }
+ TotalCost: Double { get }
+ UseWorldDistance: Boolean
+ .ctor(IGraph2D<T>)
+ FindBestPath(IGraphAgent<T>, T, T): Boolean
+ GetLastNode(Double): T
+ GetLastPathNode(Double): PathNode<T>
PathNode<T>
+ Children: ListEx<PathNode<T>> { get }
+ F: Double { get }
+ G: Double { get }
+ H: Double { get }
+ Node: T { readOnly }
+ Parent: PathNode<T> { get }
«interface»
IGraphAgent<T>
+ RelaxedRange: Boolean { get }
+ CanMakeStep(T, T): Boolean
+ CanOccupy(T): Boolean
+ GetStepCost(T, T): Double
+ IsNearTarget(T, T, Double): Boolean
Agent
ICloneable, IGraph2D<PointI>
Geometry.PolygonGrid
Graph
Agent
Coverage<T>
+ Agent: IGraphAgent<T> { get }
+ Graph: IGraph2D<T> { readOnly }
+ Nodes: ListEx<T> { get }
+ .ctor(IGraph2D<T>)
+ FindReachable(IGraphAgent<T>, T, Double): Boolean
Graph
«interface»
IGraph2D<T>
+ Connectivity: Int32 { get }
+ NodeCount: Int32 { get }
+ Nodes: IEnumerable<T> { get }
+ Contains(T): Boolean
+ GetDistance(T, T): Double
+ GetNearestNode(PointD): T
+ GetNeighbors(T): IList<T>
+ GetWorldLocation(T): PointD
+ GetWorldRegion(T): PointD[]
Graph
FloodFill<T>
+ Graph: IGraph2D<T> { readOnly }
+ Nodes: ListEx<T> { get }
+ .ctor(IGraph2D<T>)
+ FindMatching(Predicate<T>, T): Boolean
Graph
NodeArc
+ Distance: Double { readOnly }
+ Start: Double { readOnly }
+ Sweep: Double { readOnly }
+ VisibleFraction: Double { get }
+ IsObscured(NodeArc): Int32
NodeArcs
Visibility<T>
+ Graph: IGraph2D<T> { readOnly }
+ NodeArcs: DictionaryEx<T, NodeArc> { get }
+ Nodes: ListEx<T> { get }
+ Threshold: Double { get, set }
+ .ctor(IGraph2D<T>)
+ FindVisible(Predicate<T>, T, Double = 0): Boolean
ICloneable, IGraph2D<PointD>
Geometry.Subdivision

Figure 5.1: Graph Types

index of a grid element. The `Subdivision` node type is `PointD`: each graph node is the two-dimensional location of a subdivision vertex.

The following `IGraph2D<T>` members establish the nodes and edges of a graph.

`Connectivity`	— The maximum number of immediate neighbors of any graph node. A node's immediate neighbors are the nodes that share an edge with that node, and therefore can be reached within a single movement step.
`NodeCount`	— The total number of nodes in the graph.
`Nodes`	— Enumerates all nodes in the graph.
`Contains`	— Determines whether the graph contains a specified node.
`GetNeighbors`	— Finds all immediate neighbors of a specified graph node. `PolygonGrid` adds an overload that also finds all remoter neighbors within a given range of movement steps.

5.1.2 World Coordinates

`IGraph2D<T>` represents a purely abstract system of node connections, but each graph node also maps to coordinates and regions in two-dimensional space. We refer to these coordinates as "world coordinates" to distinguish them from node coordinates in some graph-specific system, e. g. the two-dimensional integer indices of `PolygonGrid` elements. In the case of `Subdivision` graphs, the two systems are identical: graph coordinates equal world coordinates.

The following `IGraph2D<T>` members establish relationships between graph nodes and world coordinates (although this is not necessarily the case for `GetDistance`).

`GetDistance` — Computes the distance between two specified graph nodes in terms of some arbitrary distance measure, which may or may not correspond to world coordinates. Generally, an implementation should use the simplest calculation that obeys two invariants. One is the step cost invariant for the associated graph agent (see below). The other requires that the sum of the distances between all successive nodes within a sequence is never less than the distance between any two nodes from the same sequence.

`PolygonGrid` counts intervening nodes, i. e. the minimum number of movement steps between immediate neighbors when moving from the source node to the target node. `Subdivision` calculates the Euclidean distance in world coordinates.

`GetNearestNode` — Finds the graph node nearest to the specified world coordinates.

`GetWorldLocation` — Gets the world location of a specified graph node. `PolygonGrid` returns the center of the polygonal element that represents the graph node. `Subdivision` simply reflects the input.

`GetWorldRegion` — Gets the vertices, in world coordinates, of the polygonal region covered by a graph node. `PolygonGrid` returns the vertices of the polygonal element that represents the graph node. `Subdivision` requires that users manually assign regions to graph nodes, e. g. Voronoi regions if the subdivision was created from the corresponding Delaunay triangulation.

5.1.3 Moving Agents

The interface `IGraphAgent<T>` represents a moving "agent," i. e. anything that can move from one graph node to another. The `AStar<T>` and `Coverage<T>` algorithms require an instance of this interface to determine valid movement steps, the cost of each step, and other data.

There is no default implementation for `IGraphAgent<T>` as the behavior of moving agents is specific to each individual application. See below for tips on implementing this interface.

`CanMakeStep` — Determines whether the agent can move from one specified graph node to another, which must be an immediate neighbor.

`CanOccupy` — Determines whether the agent can *permanently* occupy the specified graph node, i. e. whether the agent's movement path can end on that node.

`GetStepCost` — The cost for moving the agent from one specified graph node to another, which must be an immediate neighbor. The step cost can never be less than the `GetDistance` result for the two nodes. Together with the invariant regarding distances within node sequences (see above), the step cost invariant allows the `AStar<T>` algorithm to use `GetDistance` to establish lower bounds for possible path costs.

`IsNearTarget` — Determines whether the specified node is close enough to the ultimate target node that pathfinding can successfully terminate. `CanOccupy` must succeed as well.

For example, when moving units towards an attack target, reaching the target itself is unnecessary and usually even impossible. Instead, `IsNearTarget` should succeed when the agent has reached a location from which it can attack the target.

`RelaxedRange` — Indicates whether the agent's path cost limit is strict or relaxed. This option affects the `AStar<T>` and `Coverage<T>` algorithms, as described below.

5.2 A* Pathfinding Algorithm

`AStar<T>` defines the well-known A* pathfinding algorithm. The core of the C# implementation is based on the article *Basic A* Pathfinding Made Simple* by James Matthews, published in *AI Game Programming Wisdom,* Charles River Media 2002.

`AStar<T>` finds the cheapest path, in terms of the combined cost of all movement steps, from one specified graph node to another. The path is constructed as a tree of `PathNode<T>` objects which associate a graph node with the auxiliary data required by the algorithm.

Once pathfinding is complete, the final `PathNode<T>` of the cheapest path is stored in the `BestNode` property, and the path itself can be backtracked as a sequence of `Parent` pointers. This is rather laborious, so the following properties expose the results in a more convenient way:

`Nodes` — A list of all graph nodes in the movement path, from source to target.

`TotalCost` — The total cost of the entire movement path returned in `Nodes`.

`GetLastNode` — Finds the last node in the movement path whose path cost does not exceed the specified maximum cost, and for which the agent's `CanOccupy` method succeeds. The last condition ensures that the result is valid as an intermediate stop in multi-turn movements.

These properties are also grouped into a separate interface, `IGraphPath<T>`, which `AStar<T>` implements. This interface was designed to represent graph paths without dependence on any particular pathfinding algorithm, but A* is the only algorithm available so far.

In the rest of this section we'll describe two options exposed by `AStar<T>` itself to customize pathfinding, and how `IGraphAgent<T>` interacts with the pathfinding algorithms.

5.2.1 Limited Search Range

Set the floating-point property `AStar<T>.RelativeLimit` to limit the search range during pathfinding. This may cause A* to generate suboptimal paths or even fail to find any path at all, but performance on large graphs will be greatly improved.

`RelativeLimit` defaults to zero. A positive value limits all candidate paths to an elliptical area around the source and target node of the search. Any candidate nodes that would lead beyond this area are ignored. All distances are calculated using the graph's `GetDistance` method.

`RelativeLimit` determines the radii of the ellipse, relative to the distance of the source and target nodes. After a path search, the read-only property `AbsoluteLimit` holds the maximum number of movement steps that were considered for any candidate path.

5.2.2 Minimal World Distance

Set the boolean property `AStar<T>.UseWorldDistance` to eliminate zero-cost oscillations in the returned path. Such oscillations have no effect on the total path cost, which is guaranteed to be optimal, but might cause strangely "unnatural" unit movements.

This effect can occur on graphs such as `PolygonGrid` whose `GetDistance` method does not use world coordinates but some more abstract measure (in this case, the number of movement steps) which may not assign the smallest path cost estimate to the visually most direct path. The effect is pronounced on square grids with diagonal neighbor connections: rather than moving in a direct line, an agent might "sidestep" to an adjacent row or column and then back again.

`UseWorldDistance` defaults to `false`. The value `true` adds an extra comparison to decide between candidate nodes that have equal path costs. Rather than always selecting the first node that happens to be generated, A* also checks the world distance of each candidate node to the target location, and selects the node with the smallest distance.

5.2.3 Transient and Permanent Occupation

A* constructs movement paths from a sequence of individual movement steps between graph nodes that are immediate neighbors. For each step, we must ask two questions: can the moving agent make the step, and how much does it cost? The first question is answered by the two agent methods `CanMakeStep` and `CanOccupy`, the second by `GetStepCost` (described below).

We use two methods to determine whether a movement step is possible because we want to distinguish between *transient occupation* and *permanent occupation*. `CanMakeStep` performs the fundamental test whether the agent can move between the specified nodes at all, i. e. whether the source-target step can be even a transient part of its movement path. A target node for which `CanMakeStep` fails is never part of a path, unless we reach it by a different source node.

`CanOccupy` represents an additional test whether the agent can end its movement path on the specified node, and thus "permanently" occupy the node for the time being. A* requires that `CanOccupy` succeeds for the final path node, and also for any intermediate nodes returned by `GetLastNode` since they may represent intermediate stops during multi-turn movements.

This distinction between transient and permanent occupation is common in traditional board games where pieces can jump over occupied squares but land only on free squares. War games might also relax stacking limits for tiles that units only pass through.

Occupying Intermediate Nodes

A* never calls `CanOccupy` on the intermediate nodes of a path, only on the final node. This is the desired and necessary behavior. However, this can cause problems for multi-turn movements with a non-trivial `CanOccupy` implementation. Because `CanOccupy` has not been checked for intermediate nodes, `GetLastNode` might return suboptimal nodes, or none at all, when invoked with less than the total path cost.

For this reason, you must check every `GetLastNode` call for a valid result, and even if the result is valid your partial path might look rather strange. The best advice is to avoid implementations where this is a major issue. The second-best advice is to use `Coverage<T>` and heuristics to manually piece together valid movement paths through difficult environments.

5.2.4 Movement Step Costs

The total cost of a movement path found by `AStar<T>` and `Coverage<T>` is defined as the sum of the costs of all movement steps between consecutive path nodes. Step costs depend only on the moving agent and the two directly involved nodes, never on any other nodes in the same path. This assumption is fundamental since the A* algorithm constructs an optimal path from path fragments that were originally found as parts of different search paths.

The agent's integer function `GetStepCost` determines the cost of one movement step from a specified graph node to one of its immediate neighbors. This cost must be positive and no less than the graph's `GetDistance` result for all possible movement steps. `CanMakeStep` (and also `CanOccupy` for the final node of a path) is always called before `GetStepCost` to ensure that the agent can enter the target node at all.

5.2.5 Relaxed Movement Range

Implement the agent's boolean property `RelaxedRange` to indicate whether the moving agent enjoys an *extended movement range.*

If `RelaxedRange` is `false`, the maximum path cost supplied to A* limits the agent's range *before* a step is taken. If the cost of entering another location exceeds the remaining fraction of the maximum cost, the agent cannot enter. Note that this might lead to situations where an agent cannot move at all because all surrounding nodes exceed the maximum path cost.

If `RelaxedRange` is `true`, a movement path ends only *after* its total cost equals or exceeds the maximum path cost. As long as this has not happened, the agent can enter *any* neighboring node, regardless of the actual cost of this step. This means that the agent can always make at least one step in any direction, regardless of its cost.

Marking Nodes as Impassable

Assume you wish to prevent an agent from entering certain graph nodes, for example because they represent impassable terrain.

If `RelaxedRange` is `false`, you could return very high step costs for the desired nodes in your `GetStepCost` implementation. If the step costs exceed any maximum path cost supplied to the pathfinding algorithms, the agent cannot enter these nodes.

However, this trick no longer works if `RelaxedRange` is `true`. In this case, your `CanMakeStep` implementation must return `false` for the desired nodes to make them impassable.

5.3 Path Coverage Algorithm

`Coverage<T>` defines a path coverage algorithm whose results are compatible with `AStar<T>`. This algorithm finds all graph nodes that can be reached from a specified node within a given maximum path cost.

When running on the same `IGraph2D<T>` and `IGraphAgent<T>` instances, `Coverage<T>` produces exactly those target nodes for which A* would find a path, given the same or a lower maximum cost. `Coverage<T>` does not store the actual paths, however – you must run `AStar<T>` on any found target nodes for which you wish to obtain a movement path.

`Coverage<T>` uses the `IGraphAgent<T>` interface in the same way as `AStar<T>`. Note that since all graph nodes found by `Coverage<T>` represent *end points* of possible A* movement paths, the agent's `CanOccupy` implementation must succeed for all of them. Intermediate nodes of possible paths that do not allow permanent occupation will not appear in the result set.

5.4 Flood Fill Algorithm

`FloodFill<T>` defines a flood fill algorithm for arbitrary graphs that works like the eponymous function in paint programs. This algorithm finds all immediate neighbors of a specified graph node for which a given predicate succeeds, then recursively all neighbors of those neighbors and so on. The search ends when the graph is exhausted or the predicate fails for all remaining neighbors of the result nodes.

`FloodFill<T>` is essentially a simpler version of `Coverage<T>` that uses a boolean predicate instead of a full-fledged `IGraphAgent<T>` instance. Therefore, its results are not necessarily related to any valid agent movements.

5.5 Visibility Algorithm

`Visibility<T>` defines a line-of-sight algorithm that operates on a graph's world coordinates. The algorithm requires a source node and a maximum world distance from that source, as well as a predicate that determines whether a specified graph node obstructs visibility.

Currently, occlusion is binary only – a given node is considered either completely opaque or completely transparent. A node's visibility is determined as follows:

1. The node is assigned a *tangential arc,* determined by drawing tangents from the location of the source node (as per `GetWorldLocation`) to the extreme vertices of its polygonal world region (as per `GetWorldRegion`).

2. The node is assigned a *source distance,* measured from the location of the source node to the nearest vertex of its polygonal world region.

3. The node is compared against all opaque nodes that are not completely obscured by other opaque nodes. If the node's tangential arc overlaps that of an opaque node with a smaller source distance, then the overlapping fraction is considered obscured.

4. The node is considered visible from the source exactly if a certain minimum fraction of its tangential arc remains visible after comparing it against all opaque nodes.

This fraction defaults to 1/3 but can be changed to any value between zero and one by setting the `Threshold` property. Zero is equivalent to `Double.Epsilon`, i. e. a node is considered visible if even the slightest bit of its tangential arc remains unobscured. Conversely, a threshold of one requires that a visible node's tangential arc is not obscured anywhere.

The computed data for all visited nodes – tangential arc, visible fraction, and source distance – is available in the `NodeArcs` collection. Applications can use this information to fine-tune their own concept of node visibility.

Chapter 6

IO Namespace

The namespace `Tektosyne.IO` contains just a few helper classes for file input/output. They only require the four “core” BCL assemblies, and therefore reside in assembly `Tektosyne.Core`.

`IOUtility` — Static methods to search directories and directory trees for file names containing wild cards, and to textually replace specified inclusion tags in a given text file with other files.

`PathEx` — Static methods to shorten and compare file paths, manipulate directory separators (either variant), and to create random temporary files with a given extension.

`RootedPath` — Combines a fully qualified file path with an optional directory prefix, so that the file path is automatically shortened to a relative path if the prefix matches.

Chapter 7

Net Namespace

The namespace `Tektosyne.Net` contains several types for using the Simple MAPI protocol. Since this protocol is exclusive to Windows, all types reside in assembly `Tektosyne.Windows`.

`MapiAddress`	—	The sender or recipient of a Simple MAPI message.
`MapiException`	—	Thrown when an error occurs while using the Simple MAPI protocol.
`MapiMail`	—	Static methods that use Simple MAPI to access the user's address book and send e-mails with optional attachments.

The BCL has no built-in support for Simple MAPI. We access this protocol through our own managed/unmanaged interfaces defined in the namespace `Tektosyne.Win32Api`.

Chapter 8

Win32Api Namespace

The namespace `Tektosyne.Win32Api` contains import declarations for unmanaged Win32 API functions and managed representations of Win32 API structures. Naturally, all types are specific to Windows and reside in assembly `Tektosyne.Windows`.

The import declarations are by no means exhaustive. They cover only a handful of functions that are needed by other Tektosyne types, or that I used elsewhere at some point.

`Kernel`	— Interfaces the Windows system library `kernel32.dll`.
`MemoryStatus(Ex)`	— Contains information about the current state of both physical and virtual memory.
`User`	— Interfaces the Windows system library `user32.dll`.

8.1 Safe Memory Handles

The following types derive from the BCL class `SafeHandle` to provide exception-resistant handles for unmanaged memory blocks.

`SafeMemoryHandle`	— Extends `SafeHandle` with methods that copy data to and from unmanaged memory.
`SafeGlobalHandle`	— Implements `SafeMemoryHandle` for memory blocks that are allocated by `AllocHGlobal` and released by `FreeHGlobal`.
`SafeMapiHandle`	— Implements `SafeMemoryHandle` for memory blocks that are implicitly allocated by Simple MAPI functions and released by `MAPIFreeBuffer`.

8.2 Simple MAPI Protocol

The following types interface the Simple MAPI protocol. The namespace `Tektosyne.Net` builds convenient high-level wrappers around these types.

`Mapi`	— Interfaces the Windows system library `mapi32.dll`.
`MapiFileDesc`	— Contains information about a file containing a message attachment stored as a temporary file. (Seriously, this is the actual summary from the Windows SDK documentation!)
`MapiFileTagExt`	— Specifies a message attachment's type at its creation and its current form of encoding so that it can be restored to its original type at its destination.
`MapiMessage`	— Contains information about a Simple MAPI message.
`MapiRecipDesc`	— Contains information about a message sender or recipient.
`MapiError`	— Defines error codes returned by Simple MAPI calls.
`MapiFileFlags`	— Defines attachment flags for MAPI messages.
`MapiFlags`	— Defines flags supplied to Simple MAPI calls.
`MapiMessageFlags`	— Defines status flags for MAPI messages.
`MapiRecipClasses`	— Defines recipient classes for MAPI messages.

Chapter 9

Windows Namespace

The namespace `Tektosyne.Windows` contains types that support the Windows Presentation Foundation (WPF), and naturally resides in assembly `Tektosyne.Windows`.

Many `Tektosyne.Windows` types are unrelated except in their general purpose to simplify WPF development; these are listed below. The following sections group the remaining types which handle bitmap manipulation, concurrent drawing, default theme selection, and Windows Forms interoperability.

`ContainerVisualHost` — Extends `FrameworkElement` to host a single `ContainerVisual` child element. This lets you host `Visual` objects in structures that require a `UIElement` or `FrameworkElement`.

`FormatTextBlock` — Extends `TextBlock` with persistent format specifications and formatting methods.

`MessageDialog` — Extends `Window` to simulate a `MessageBox` with a scrollable secondary message area.

`StackTextBlock` — Extends `TextBlock` with a stack of recently displayed messages, so that clients can easily restore a previous message without having to store it locally.

`TaskEvents` — Combines completion, notification, and error events with a `Stopwatch` timer to simplify task management. All events are marshalled across a specified `Dispatcher` if desired.

`TreeHelper` — Static methods to simplify navigating the logical and visual trees of WPF objects, including a method that finds the owner of an active `ContextMenu` from its clicked item.

`WindowsExtensions` — Static methods that extend various WPF types to process a `Dispatcher`'s message queue; get a `Control`'s type face; scroll a `ScrollViewer` by a

	specified offset; ensure that a selected `ListBox` item is visible; and convert between GDI+ and WPF `Color` values.
`WindowsUtility`	— Static properties and methods that retrieve the system's memory status; determine the DPI resolution of a `Visual` or the primary screen; and find the `Icon` or `BitmapSource` for a `MessageBoxImage` value.
`ScrollDirection`	— Specifies the four possible scroll directions. Used by `WindowsExtensions` methods.

9.1 Bitmap Manipulation

This section comprises types for directly manipulating the pixels of an SRGB bitmap, which is stored in an instance of the WPF `WriteableBitmap` class.

`BitmapBuffer`	— Provides a secondary buffer for a `WriteableBitmap`. Prior to .NET 3.5 SP1, such a buffer was necessary for accessing the bitmap's pixels.
`BitmapUtility`	— Static methods to access the `BackBuffer` pixels of a `WriteableBitmap`. As of .NET 3.5 SP1, this is faster and more convenient than using a separate `BitmapBuffer`.
`ColorVector`	— Represents a displacement in SRGB color space which can be added to `Color` values.

9.2 Concurrent Drawing

WPF drawing always has some inherent concurrency, due to the separation of *dispatcher thread* and *render thread.* The helper classes described in this section increase concurrency by running *multiple dispatcher threads* within the same window. However, the single render thread still places an upper limit on the amount of parallelism achievable purely within WPF.

Let's examine each of these rather complex issues in more detail.

9.2.1 Dispatcher & Render Thread

All user code in a WPF application normally lives on a single dispatcher thread. This is the thread that calls `Application.Run` to start the `Dispatcher` loop, which keeps processing user input and other Windows messages throughout the application's lifetime. All drawing starts on this thread, either implicitly by placing WPF elements that have a predefined appearance, or explicitly by calling drawing methods within an `OnRender` override etc.

Once the dispatcher thread has processed all drawing instructions and returned to its input loop, the prepared drawing content is handed off to the render thread. This thread always runs in the background of a WPF application and is completely inaccessible to user code. Its job is to translate the prepared content into actual screen pixels, using the DirectX API.

9.2.2 Multiple Dispatcher Threads

WPF provides two well-known options for concurrent drawing. You can create "frozen" objects in a regular background thread, or you can associate multiple top-level windows with separate dispatcher threads. However, WPF team member Dwayne Need has described a third mechanism that allows multiple threads to draw mutable objects within the same window.

This option combines several rather arcane WPF classes (`HostVisual`, `PresentationSource`, `VisualTarget`) to associate an arbitrary `Visual` with its own `Dispatcher` running on a background thread. The Tektosyne library provides a simplified wrapper for this task.

`ConcurrentVisualHost` — This is the only class you need. An instance can be placed anywhere a `FrameworkElement` is acceptable. The `WorkerDispatcher` and `WorkerVisual` properties expose the `Dispatcher` and `Visual`, respectively, that live on an automatically started background thread. You may specify a `WorkerVisual` during construction, but you can also change it at any time.

`VisualSource` — Manages cross-thread marshalling within a `ConcurrentVisualHost`. This class was adapted from Dwayne Need's example. You don't need to instantiate it yourself, unless you wish to write your own version of `ConcurrentVisualHost`.

For an example of how to use `ConcurrentVisualHost`, please refer to the source code for the `ConcurrentDrawingTest` dialog in the demo application `Tektosyne.GuiTest.exe`. The following points are especially noteworthy:

— All `WorkerVisual` operations must be wrapped in `WorkerDispatcher.BeginInvoke` calls. The dispatcher invocation is necessary because `WorkerVisual` lives on a background thread, and `BeginInvoke` rather than `Invoke` is necessary to achieve concurrent drawing.

— `ConcurrentVisualHost` implements `IDisposable` to allow manually stopping `WorkerDispatcher` and terminating its background thread. Call `Dispose` when removing a `ConcurrentVisualHost` from an application that will continue running.

— Do not directly embed `ConcurrentVisualHost` in a XAML page! Visual Studio attempts to instantiate all declared objects when a XAML page is opened in design view. This starts unneeded background threads and can even freeze Visual Studio for several seconds. Always create and attach `ConcurrentVisualHost` objects from code-behind.

9.2.3 Limitations

As it turns out, multi-threaded drawing in WPF is not quite as awesome as one might think. There are two serious limitations. Presumably these are also the reason why Microsoft does not provide a standard class with the functionality of `ConcurrentVisualHost`.

1. Background dispatcher threads do not receive user input. If users should be able to interact with visual elements owned by a background thread, you must capture mouse clicks on the foreground thread and perform hit detection manually.

2. Background dispatcher threads only speed up *dispatcher* operations, not *render thread* operations. There is still only a single render thread, so you won't see much of a speedup if your content is more expensive to render to the screen (on the render thread) than to prepare for drawing (on the dispatcher thread).

The second problem surfaced in the `ConcurrentDrawingTest` dialog. An early version created thousands of random rectangles in each worker thread. As it turned out, the render thread took so long to process so many shapes that all measurable speedup was lost! The current version "cheats" by drawing just 100 rectangles and calling `Thread.Sleep(1000)`. This simulates drawing operations that are expensive on the dispatcher thread but cheap on the render thread.

What are your options if an overworked render thread prevents decent concurrency? First, you could try moving more work into the dispatcher thread, e. g. by manually clipping hidden objects so that drawing operations without visible effect are not emitted in the first place.

Second, you could draw to a `RenderTargetBitmap` and show that bitmap instead of executing your drawing operations directly on the `WorkerVisual`. However, if that is acceptable you might consider creating the bitmap on an ordinary `ThreadPool` thread instead, and simply freeze it for display.

Lastly, if nothing else helps you'll have to use a different API that does its own rendering, e. g. an embedded SlimDX or SharpDX surface; but first you might wish to consult my page WPF Drawing Performance for basic tips on how to speed up custom drawing in WPF.

9.3 Default Theme Selection

Windows themes define the overall appearance of a WPF application. The BCL ships with seven default themes, matching the default themes of various Windows editions.

Usually, WPF automatically selects an appropriate default theme for the target system. The following types allow selecting a different default theme when a WPF application starts up.

`DefaultTheme`	— Enumerates the available WPF default themes.
`DefaultThemeSetter`	— Changes the application's theme to the specified `DefaultTheme`.

9.4 Windows Forms

Originally intended as the successor to Windows Forms, WPF was quietly put into maintenance mode after a long and troubled development before achieving feature parity with the older API. Now it is unclear if and when either WPF or a possible Silverlight-based successor will officially add the missing functionality. Fortunately, it's fairly simple to access existing Windows Forms controls and dialogs from a WPF application.

The following types are all related to Windows Forms, either improving existing types or helping to interface the two APIS.

`ComponentControl` — Extends a Windows Forms `Control` to automate the disposal of non-`Control` components, including error providers, help providers, and tool tips.

`CustomColorDialog` — Extends a Windows Forms `ColorDialog` to automatically persist its set of custom colors, and to accept a WPF `Window` owner and WPF `Color` values.

`HwndWrapper` — Implements a Windows Forms `IWin32Window` whose `IntPtr` handle is specified explicitly or else derived from a WPF `Window` or `IWin32Window` (an eponymous and equivalent but incompatible interface!).

This conversion wrapper allows passing WPF windows to certain Windows Forms dialogs, such as `ColorDialog`.

`NumericUpDownEx` — Extends a Windows Forms `NumericUpDown` control to automatically validate manually entered values, show a tool tip for the legal range of values, and show an error message if the current value is invalid.

`NumericUpDownHost` — Extends `ComponentControl` to conveniently host one `NumericUpdownEx` control, along with its dedicated tool tip and error provider components. Forwarders provide direct access to the properties of the hosted control.

`WindowsFormsHostEx` — Extends a WPF `WindowsFormsHost` to correctly transfer input focus to the hosted control when the corresponding `Target` binding of a WPF `Label` is activated. Without this help, WPF seems to ignore all Windows Forms controls in a WPF `Window` but the first.

9.4.1 Hosting Example

Let's see how we can employ these types to host a `NumericUpDownEx` control in a WPF window. Figure 9.1 shows the complete XAML declaration. No code-behind is necessary, except to handle the `ValueChanged` event. Observe the following in particular:

— The label's `Target` refers to the `WindowsFormsHostEx` element, not to the embedded `NumericUpDownHost`. This enables the host to correctly transfer input focus.

— The hosted Windows Forms control, `NumericUpDownHost`, is simply specified as the single child element of the `WindowsFormsHostEx` element.

— We use `NumericUpDownHost` instead of `NumericUpDown(Ex)` so that the associated tool tip and error provider components are correctly disposed of.

The `NumericUpDownHost.Name` property merely allows the attached `ValueChanged` handler to access the current input value, and is not used by other XAML elements.

The involved `Tektosyne.Windows` types are shown in Figure 9.2. The forwarder properties of `NumericUpDownHost` allow customizing the embedded `NumericUpDownEx` control directly from XAML. The latter is implicitly created and does not appear in the XAML declaration.

```
<Window x:Class="ChangeSidesDialog"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:tw="clr-namespace:Tektosyne.Windows;assembly=Tektosyne.Windows">
  <StackPanel Orientation="Horizontal">
    <Label Target="{Binding ElementName=SidesUpDownHost}">_Sides:</Label>
    <tw:WindowsFormsHostEx x:Name="SidesUpDownHost" Width="60">
      <tw:NumericUpDownHost x:Name="SidesUpDown"
        Minimum="3" Maximum="12" TextAlign="Right"
        ValueChanged="OnSidesChanged" />
    </tw:WindowsFormsHostEx>
  </StackPanel>
```

Figure 9.1: `NumericUpDown` Hosting

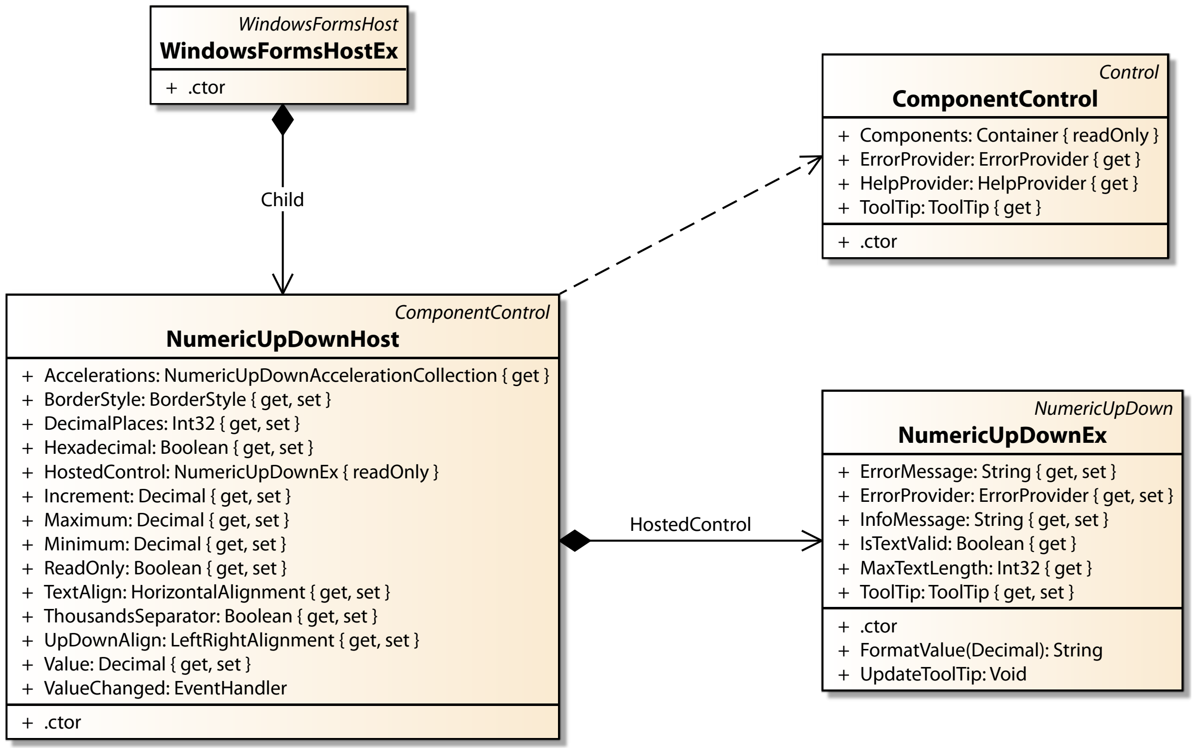
WindowsFormsHost
WindowsFormsHostEx
+ .ctor
Child
Control
ComponentControl
+ Components: Container { readOnly }
+ ErrorProvider: ErrorProvider { get }
+ HelpProvider: HelpProvider { get }
+ ToolTip: ToolTip { get }
+ .ctor
ComponentControl
NumericUpDownHost
+ Accelerations: NumericUpDownAccelerationCollection { get }
+ BorderStyle: BorderStyle { get, set }
+ DecimalPlaces: Int32 { get, set }
+ Hexadecimal: Boolean { get, set }
+ HostedControl: NumericUpDownEx { readOnly }
+ Increment: Decimal { get, set }
+ Maximum: Decimal { get, set }
+ Minimum: Decimal { get, set }
+ ReadOnly: Boolean { get, set }
+ TextAlign: HorizontalAlignment { get, set }
+ ThousandsSeparator: Boolean { get, set }
+ UpDownAlign: LeftRightAlignment { get, set }
+ Value: Decimal { get, set }
+ ValueChanged: EventHandler
+ .ctor
HostedControl
NumericUpDown
NumericUpDownEx
+ ErrorMessage: String { get, set }
+ ErrorProvider: ErrorProvider { get, set }
+ InfoMessage: String { get, set }
+ IsTextValid: Boolean { get }
+ MaxTextLength: Int32 { get }
+ ToolTip: ToolTip { get, set }
+ .ctor
+ FormatValue(Decimal): String
+ UpdateToolTip: Void

Figure 9.2: `NumericUpDown` Types

Chapter 10

Xml Namespace

The namespace `Tektosyne.Xml` contains helper methods for XML serialization. The BCL defines several different APIs for this purpose, and only some of them reside in the “core” assemblies. Consequently, our types are split between `Tektosyne.Core` and `Tektosyne.Windows`.

`XmlUtility`	— Static helper methods to manipulate and read typed attributes from an `XmlReader`. Resides in assembly `Tektosyne.Core`.
`XmlSerialization`	— Static helper methods to read an `XAttribute` as an enumeration value, and to (de)serialize objects using `NetDataContractSerializer`. Resides in assembly `Tektosyne.Windows`.